Snapshots Generation via Constructive Logic

- Extended Abstract -

Mario Ornaghi, Camillo Fiorentini and Alberto Momigliano

Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
{ornaghi,fiorenti,momiglia}@dsi.unimi.it

1 Introduction

A software information system *S* allows users to store, retrieve and process information about the external world, typically a data base. We can differentiate two separate aspects in the data elaborated by *S*: the first concerns *data types*, while the second is related to the *information on the external "real world"* carried by the data. More precisely, a data type is a set of data together with the associated manipulations where the focus is on *operations*. In contrast, the other information carried by the data stored in *S* is strongly related to their *meaning in the real world*. The need for properly treating data according to their meaning is becoming increasingly important, due to the wide quantity of information that is exchanged on the Internet [10].

In this context, we are developing COOML [9] (Constructive Object Oriented Modeling Language), an OO specification language where the focus is on the information carried by data. The main features of COOML are: a data model based on a predicative extension of the constructive intermediate logic E^* [8], based on the BHK interpretation of the logical connectives [11]; a multi-layered structure, namely with one for the COOML logic layer, others for the problem logic and the computations; the possibility of choosing different problem logics and implementation languages. The data model is based on the notion of "pieces of information" $i : S$, where i is a structured information ("information value") giving constructive evidence for the truth of a specification S. Pieces of information support both interoperability in heterogeneous systems and multiple views on data in heterogeneous knowledge domains.

We believe that COOML may have a role as an OO specification framework, and we are developing different prototypical tools. In this abstract, we focus on automatic snapshots generation. In modeling languages, a snapshots represents a system state. In particular, in the context of the UML, snapshots are object diagrams and snapshots generation in the presence of OCL constraints has proved to be useful both for understanding a specification and verifying its consistency in the problem domain [4]. In our approach, our snapshots generation algorithm is driven by the constructive content derived from the the computation of information values.

2 COOML Overview

Here we briefly discuss the COOML logic and the problem domain layers. More details can be found in [9]. The link between the data stored in a software system and their *meaning* in the "real world" is the result of the abstractions performed in the *analysis phase*. Typically (see e.g., [6]), the analysis has to produce a *dictionary* containing the abstract concepts used

in specifications and choose the *data types*. The analysis phase should result in a language to *talk about the world and its states*. We call *problem formulas* the sentences of this language. World-states can be formalized as classical interpretations. However, other problem logics can be used, and even informal interpretations are allowed. We only need to assume that problem formulas can be understood by the final user as properties that may or may not *hold in a world-state w*. In problem formulas we may have expressions related to the implementation language. Since we are interested in an OO approach, COOML introduces a specific syntax to deal with objects and classes. There is a special predefined data type *Obj* for *object-identities*. *Class predicates* are of the form $o.C()$, holding in a world-state w iff o is a *live object* of w. Since objects are rarely isolated entities, we also use class predicates of the form $o.C(e_1,\ldots,e_n)$, linking an object o of class C to its external environment by means of the *environment variables* $e_1,\ldots,e_n$.

Example 1. We consider a simple "cash-register problem domain". The dictionary provides class predicates such as $c.CashRegister()$ (c is a cash-register), $r.Receipt(c)$ (r is the receipt of the cash-register c), $i.Item(r)$ (i is an item of the receipt r), and introduces terms such as *cost, code, price of an item*, and *grand total of a receipt* (with self-explanatory meaning). If that is the case, we can formalize some properties of the problem domain. For example, we can give the following axioms, where *grandtotal* and *price* are partial functions:

$$\begin{array}{lll}
(C1)\ r1 = r2 & \leftarrow & it.Item(r1) \wedge it.Item(r2) \\
(C2)\ grandtotal(r) = \Sigma_{i.Item(r)} price(i) & \leftarrow & r.Receipt(c) \\
(C3)\ price(i) = cost(i) + cost(i) * VAT/100 & \leftarrow & i.Item(r)
\end{array}$$

The COOML logical layer takes the problem domain formulas as *atoms*, while reasoning on them is delegated to the problem domain logic. The syntax of a COOML specification P is shown below, where $\underline{\tau}\ \underline{x}$ denotes a list of distinct variables of appropriate type $\underline{\tau}$ and similarly for constants $\underline{c} : \underline{\tau}$.

$$\begin{array}{lll}
AT & ::= & PF \mid \Box P \\
P & ::= & AT \mid \text{AND}\{P \ldots P\} \mid \text{OR}\{P \ldots P\} \mid \text{EXI}\{\underline{\tau}\ \underline{x} : P\} \mid \text{FOR}\{\underline{\tau}\ \underline{x} \mid G(\underline{x}) : P\}
\end{array}$$

COOML's *atoms (AT)* consist of arbitrary problem formulas PF and $\Box$-formulas. The latter serve the purpose of embedding classical truth in our constructive setting. In our language, as in JML and OCL, universal quantification is bounded The *generator* $G(\underline{x})$ is a particular *problem formula*, true for finitely many ground instances $\underline{c}_1,\ldots,\underline{c}_m$ of closed terms; we call them the *terms generated by G(x)*.

We now address the semantics of atoms. The only information on a world-state w carried by a ground problem formula $A\sigma$ is the elementary information value *true*. It simply means that $A\sigma$ *holds in w*. The truth of a $\Box$-formula is defined interpreting the COOML connectives as ordinary logical ones.

For a specification P, we define the *information type* $\text{IT}(P)$ of P as follows, where information values are lists built starting from the primitive data types of the problem domain:

$$\begin{array}{rcl}
\text{IT}(A) & = & \{true\}, \text{ where } A \text{ is an AT} \\
\text{IT}(\text{AND}\{P_1\ \ldots\ P_n\}) & = & \{(i_1,\ldots,i_n) \mid i_j \in \text{IT}(P_j), 1 \le j \le n\} \\
\text{IT}(\text{OR}\{P_1\ \ldots\ P_n\}) & = & \{(k,i) \mid 1 \le k \le n \text{ and } i \in \text{IT}(P_k)\} \\
\text{IT}(\text{EXI}\{\underline{\tau}\ \underline{x} : P\}) & = & \{(\underline{c}, i) \mid \underline{c} : \underline{\tau} \text{ and } i \in \text{IT}(P)\} \\
\text{IT}(\text{FOR}\{\underline{\tau}\ \underline{x} \mid G : P\}) & = & \{((\underline{c}_1, i_1),\ldots,(\underline{c}_m, i_m)) \mid \\
 & & m \ge 0 \text{ and, for } 1 \le j \le m,\ \underline{c}_j : \underline{\tau} \text{ and } i_j \in \text{IT}(P)\}
\end{array}$$

A specification P gives meaning to the information values that belong to $\text{IT}(P)$. A *piece of information* is a pair $i : P$, with $i \in \text{IT}(P)$. For every ground substitution σ, the *meaning of*

$i : P\sigma$ in a world-state w is given by the relation $w \Vdash i : P\sigma$ ($i : P\sigma$ *is true in* w):

$$\begin{array}{rll}
w \Vdash \mathit{true} : A\sigma & \text{IFF} & A\sigma \text{ holds in } w, \text{ where } A \text{ is an AT} \\
w \Vdash (i_1,\ldots,i_n) : \text{AND}\{P_1 \ \ldots \ P_n\}\sigma & \text{IFF} & w \Vdash i_j : P_j\sigma, \text{ for all } j = 1,\ldots,n \\
w \Vdash (k,i) : \text{OR}\{P_1 \ \ldots \ P_n\}\sigma & \text{IFF} & w \Vdash i : P_k\sigma \\
w \Vdash (\underline{c},i) : \text{EXI}\{\underline{\tau}\,\underline{x} : P(\underline{x})\}\sigma & \text{IFF} & w \Vdash i : P(\underline{c})\sigma \\
w \Vdash L : \text{FOR}\{\underline{\tau}\,\underline{x} \mid G(\underline{x}) : P(\underline{x})\}\sigma & \text{IFF} & (\underline{c} \in dom(L) \text{ iff } G(\underline{c})\sigma \text{ holds in } w) \\
 & & \text{and } ((\underline{c},i) \in L \text{ entails } w \Vdash i : P(\underline{c})\sigma)
\end{array}$$

For example, let us consider a piece of information in the cash-register system. The information contained is: the receipt `r7` has two items, `it1` and `it2`, having costs 10 and 5 respectively.

$$((\texttt{it1}\ (\texttt{10}\ \texttt{true}))\ (\texttt{it2}\ (\texttt{5}\ \texttt{true}))) : \text{FOR}\{Obj\ it \mid it.Item(\texttt{r7}) : \text{EXI}\{\ float\ c : c = cost(i)\}\}$$

2.1 Class Specifications in COOML

In COOML we introduce classes via class predicates for the problem domain. The specification of the objects of class C is provided by a class definition of the following form, where E_C is a problem formula, S_C is a specification with name *PtyName* and M_C is a list of methods prototypes, possibly with pre and post-conditions:

Class C {
ENV$\{\ \underline{\tau}\ \underline{e} \mid this.C(\underline{e}) :\ E_C(this,\underline{e});\}$
PtyName : $S_C(this,\underline{e})$
M_C
}

The meaning of the above class specification in the COOML logic is given *constraint axioms* and *class axioms*:

$$\begin{array}{rcl}
\textbf{ConstrAx}(C) : \Box(\ \text{FOR}\{\ \text{Obj } this,\ \underline{\tau}\ \underline{e}\ \mid\ this.C(\underline{e})\ :\ E_C(this,\underline{e})\ \}) \\
\textbf{ClassAx}(C)\ \ :\ \ \ \ \text{FOR}\{\ \text{Obj } this,\ \underline{\tau}\ \underline{e}\ \mid\ this.C(\underline{e})\ :\ S_C(this,\underline{e})\ \}
\end{array}$$

Constraint axioms do not represent any information on the current world-state w; what $w \Vdash \mathit{true} : \textbf{ConstrAx}(C)$ requires is only that $\textbf{ConstrAx}(C)$ holds in w. In contrast, $\textbf{ClassAx}(C)$ allows us to represent information on the current state by means of the corresponding information values. An OO system is (specified by) a set $\mathcal{S}$ of class definitions. We associate with it the set of first order axioms $\textbf{Ax}(\mathcal{S})$ containing all axioms for all the classes in $\mathcal{S}$.

Example 2. The axioms of Ex. 1 express the general properties of the problem domain, while the following COOML classes specify the cash OO system regarding its information content.

Class CashRegister {
CashRegisterPty: OR{ EXI{ Obj !*receipt* : *receipt*.*Receipt*(*this*); }
The receipt is empty; }
}

Class Receipt {
ENV{ Obj *cash* | *this*.*Receipt*(*cash*) : *true*;}
ReceiptPty: AND{ FOR{ Obj *item* | *item*.*Item*(*this*) : *item*.*inCatalog*(); }
EXI{ *float total* : *total* = *grandtotal*(*this*) } }
}

Class Item {
ENV{ Obj *receipt* | *this.Item*(*receipt*) : *this.inCatalog*();}
ItemPty: AND{ EXI{ *float cost* : *cost* = *cost*(*this*) }
EXI{ *float price* : *price* = *price*(*this*); } }

```
/* ensures \result = cost + cost*VAT/100 */
float price();
}
```

The corresponding constraints and class axioms are defined as above; for instance, the ones for Item are:

ConstrAx(Item) : □(FOR{Obj *receipt this* | *this.Item*(*receipt*) : *this.inCatalog*(); })
ClassAx(Item) : FOR{ Obj *receipt this* | *this.Item*(*receipt*) : **ItemPty**(*this*, *receipt*)}

Assume that $\mathcal{P}_C$: **ClassAx**(C) is the piece of information of a class axiom **ClassAx**(C). In fact, $\mathcal{P}_C$ is a (possibly empty) list of information values of the form $((o\ \underline{t})\ i)$, where o instantiates *this* and $\underline{t}$ is a tuple of terms instantiating the environment variables $\underline{e}$. We call $\mathcal{P}_C$ a *population of class C* and treat it as a set. The population $\mathcal{P}$ of an OO system is the union of the populations of its classes. We say that an object o belongs to the population $\mathcal{P}$ iff there is an information value $((o\ \underline{t})\ i)$ in $\mathcal{P}$. A population $\mathcal{P}$ is finite (an OO system has a finite set of objects) and each object o of $\mathcal{P}$ occurs in a unique information value $((o\ \underline{t})\ i)$ in $\mathcal{P}$ (an object belongs to an OO system in a unique copy). We identify system states with populations, and we define the semantics of system states as follows:

Definition 1. *Let $\mathcal{P}$ be a population for a system $\mathcal{S}$ and w a world-state. Then $w \Vdash \mathcal{P} : \mathcal{S}$ iff:*

1. *$w \Vdash \mathcal{P}_C$: **ClassAx**(C) for every class C of $\mathcal{S}$, where $\mathcal{P}_C$ is the population of class C;*
2. *Every constraint axiom A of **Ax**$(\mathcal{S})$ holds in w.*

In our approach, an OO system $\mathcal{S}$ is consistent iff it has a consistent population $\mathcal{P}$, and $\mathcal{P}$ is consistent iff there is at least a world-state w such that $w \Vdash \mathcal{P} : \mathcal{S}$.

2.2 Snapshots Generation and Consistency

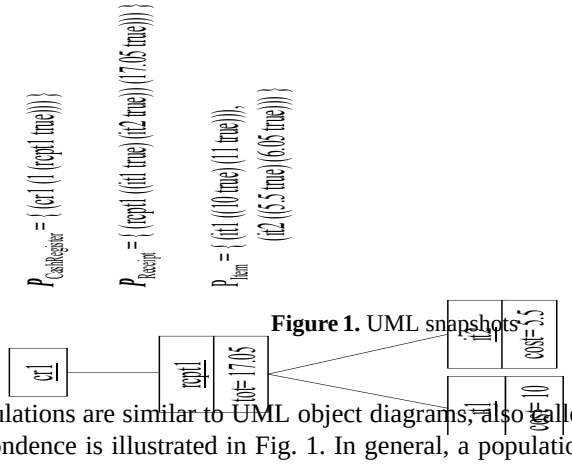

Figure 1. UML snapshots

Populations are similar to UML object diagrams, also called system snapshots [6]. This correspondence is illustrated in Fig. 1. In general, a population contains more information

than the corresponding UML snapshots and, more importantly, the pieces of information contained in it are related in a precise logical way to the problem domain. Formally:

Theorem 1. *Let $P\sigma$ be an instance of a COOML specification and w be a reachable world-state of the problem domain. Then: (a) $P\sigma$ holds in w iff there is an information value $i \in \mathrm{IT}(P)$ such that $w \Vdash i : P\sigma$; (b) for every $j \in \mathrm{IT}(P)$, we can extract a set $\mathcal{A}(j, P\sigma)$ of ground atoms from j such that $w \Vdash j : P\sigma$ iff $\mathcal{A}(j, P\sigma)$ holds in w.*

This theorem allows us to use populations for the following purposes:

Run time visualization and verbalization tools. In an implementation of COOML specifications, we can gather at run time information values for the system or a portion of it. This allows us to extract the information contained in the memory state and visualize it in a way similar to UML snapshots; moreover, we can also verbalize it in a human readable way, using the signature and the knowledge of the problem domain. This is loosely connected to the issue of *proof verbalization* from proof-terms [3].

Run time checking of problem domain properties. For that, we use the set $\mathcal{A}(i, P\sigma)$ of atoms corresponding to the extracted information values. We assume that the property to be checked is formalized in an appropriate language. For example, let us assume that the current population is as in Fig. 1. To check the constraint (C2) of Ex. 1 using logic programming for the problem logic, we extract the set $\mathcal{A}(i, \mathcal{P}_{Receipt}\sigma)$ containing the atoms `rcpt1`.*Receipt*(`c1`), `it1`.*Item*(`rcpt1`), `it2`.*Item*(`rcpt1`), 11 = *price*(`it1`), 6.05 = *price*(`it2`), 17.05 = *grandtotal*(`rcpt1`). Translating the above atoms into the required relational form, we can check the constraint (C2) using the following clause (with *sum* suitably defined):

```
false(R,T):- R.receipt(C), sum(X, (I.item(R), cost(I, X)), Sum),
             total(R,T), T =\= Sum.
```

If the goal `false(R,T)` fails, then the constraint is validated. Otherwise, we obtain the receipts where the total is wrong.

Snapshots generation and consistency checking Using the recursive definition of $\mathrm{IT}(P)$, we can generate information values for the purpose of automatic snapshots generation and property checking. Snapshots generation occurs in two stages. In the first one, we generate a partially defined information value $p(\underline{x}) \in \mathrm{IT}(P)$, where object names are instantiated and the other data are indicated by suitable variable symbols $\underline{x}$, related to the existential and universal quantifiers of P. In the second stage, the atoms $\mathcal{A}(p(\underline{x}), P\sigma)$ are extracted and their consistency is checked. Of course, in general consistency is not decidable. However we can check certain properties using one of the existing tools or formalizing the former (if possible) in a suitable Horn theory.

It is worthwhile to give some insights on the algorithm for extracting the partial information values, because this algorithm already checks some constraints on the objects of a population. The user is required to name possible objects of a class, values of an attribute, environments, and the maximum number of snapshots (information values). This could be accomplished by means of a dedicated language such as ASSL [4,5]. We represent the relationships between objects stemming from the class specification and the user inputs as certain *admit*-clauses. For example, choosing at most two cash-registers and at most two items for each receipt, with `c1`,`c2`, `it1`,`it2` constants, we get:

$$admit(x.CashRegister()) \quad \leftarrow \quad member(x, [\texttt{c1}, \texttt{c2}]).$$
$$admit(receipt.Receipt(x)) \quad \leftarrow \quad admit(x.CashRegister()).$$
$$admit(x.Item(y)) \quad \leftarrow \quad y = receipt.Receipt(x) \wedge admit(y) \wedge member(x, [\texttt{it1}, \texttt{it2}]).$$

The second *admit*-clause is fixed at the level of problem domain (each cash-register has at most one receipt). The algorithm tries to generate the information values for a class C, choosing the live objects and the attribute values according to the *admit*-clauses. The instantiation of the object names proceeds while the information value is being constructed. To this aim, algorithm maintains two lists: the list *Done* of the current completely generated objects and a list *ToDo*, containing a partially processed population. For example, if we have the lists $Done = [c1.CashRegister()]$ and $ToDo = [it1.Item(x), it2.Item(x)]$, a new partially defined object $x.Receipt(y)$ is generated, and then the solution $y = \mathtt{c1}$, $x = receipt(\mathtt{c1})$ is obtained. The list *Done* becomes

$$[\mathtt{c1}.CashRegister(), receipt.Receipt(\mathtt{c1}.CashRegister()),\\ \mathtt{it1}.Item(receipt.Receipt(\mathtt{c1}.CashRegister())), \mathtt{it2}.Item(receipt.Receipt(\mathtt{c1}.CashRegister()))]$$

and *ToDo* the empty list. We give an idea of the algorithm for generating the information values with Prolog-like pseudo-code. We only show the definition of the main predicate $info(+Formula, -Info, +Done, +ToDo, -Done', -ToDo')$

$$\begin{array}{l}
info(A,true,L,C,L,C) \;\leftarrow\; isAtom(A).\\
info(and\{F_1 \ldots F_n\},[I_1,\ldots,I_n],L_0,C_0,L_n,C_n) \;\leftarrow\; \bigwedge_{j=1}^{n} info(F_j,I_j,L_{j-1},C_{j-1},N_j,C_j).\\
info(or\{F_1 \ldots F_n\},[k,I],L,C,NL,NC) \;\leftarrow\; info(F_k,I,L,C,NL,NC). \quad k=1,\ldots,n\\
info(exi\{T\ ,F\},[V,I],L,C,NL,NC) \;\leftarrow\; choose(V,T,L,C,L1,C1) \wedge info(F,I,L1,C1,NL,NC).\\
info(for\{T\ X \mid G:F\},FI,L,C,NL,NC) \;\leftarrow\; generate(X,T,G,Dom,L,C,L1,C1) \wedge\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad map(F,Dom,FI,L1,C1,NL,NC).
\end{array}$$

The predicate $choose(V,T,L,C,L1,C1)$ chooses a value for V according to the type T[1] and the *admit*-clauses, and updates (if needed) the lists. Similarly, the predicate $generate(X,T,G,Dom,L,C,L1,C1)$ generates a domain *Dom* for the generator formula G. Finally, $map(F,Dom,FI,L1,C1,NL,NC)$ computes $[I_{v_1},\ldots,I_{v_n}]$, where $v_1,\ldots,v_n$ are the elements of *Dom*. The generation succeeds provided that $ToDo'$ is empty, and builds an $Info$ of $\mathrm{IT}(Formula)$ that has objects $Done'$.

3 Conclusion

Various logically based modeling languages of OO systems have been proposed, using different formal contexts (e.g., [2,12,1]). The setup of COOML is the constructive semantics of *evaluation forms*. This semantics is related to Medvedev's logic of finite problems [7] and it has been studied in [8]. Our general aim is to design a logically based framework for the specification of OO information systems, intended as software systems to store and manipulate information with an external meaning. COOML has been outlined in [9]. In this abstract we have considered the use of the COOML data model, based on pieces of information, for the purposes of run time visualisation and verbalisation of the information stored in the system state, run time checking of system properties, and snapshot generation.

In particular, one of the advantages of using a constructive approach to OO modeling is that the semantics itself, namely the notion of piece of information, directly supports the *declarative* generation and validation of system snapshots. Moreover, in contrast with model based approaches, information values can be shaped at different levels of granularity. For example, we can use $\mathrm{EXI}\{\tau x : \Box\mathrm{EXI}\{\tau y : r(x,y)\}\}$ to indicate that the information value has to contain a witness for x, but not for y. This opens the door to applying constructive logical methods to express and formally reason about more intensional system properties.

[1] In the internal representation, T may contain environment informations.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.
2. A. Baker, C. Ruby, and G. Leavens. Preliminary design of JML: A behavioral interface specification language for Java. Technical report, Department of Computer Science, Iowa State University, June 02 1999.
3. A. Fiedler. *P.rex*: An interactive proof explainer. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning — 1st International Joint Conference, IJCAR 2001*, number 2083 in LNAI, pages 416–420, Siena, Italy, 2001. Springer Verlag.
4. M. Gogolla, J. Bohling, and M. Richters. Validation of UML and OCL models by automatic snapshot generation. In *UML*, pages 265–279, 2003.
5. M. Gogolla, M.Richters, and J. Bohling. Tool support for validating UML and OCL models through automatic snapshot generation. In *SAICSIT '03*, pages 248–257, 2003.
6. C. Larman. *Applying UML and Patterns*. Prentice Hall PTR, Upper Saddle River, NJ, 1998.
7. J. Medvedev. Interpretation of logical formulas by means of finite problems and its relation to the realizability theory. *Soviet Mathematics Doklady*, 4:180–183, 1963.
8. P. Miglioli, U. Moscato, M. Ornaghi, and G. Usberti. A constructivism based on classical truth. *Notre Dame Journal of Formal Logic*, 30(1):67–90, 1989.
9. M. Ornaghi, M. Benini, M. Ferrari, C. Fiorentini, and A. Momigliano. A constructive object oriented modeling language for information systems. *ENTCS*, 2005. To appear.
10. A. Sheth. DB-IS research for Semantic Web and enterprises. Brief history and agenda. LSDIS Lab, Computer Science, University of Georgia, 2002. `http://lsdis.cs.uga.edu/SemNSF/Sheth-Position.doc`.
11. A. Troelstra. Aspects of constructive mathematics. In J. Barwise, editor, *Handbook of Mathematical Logic*. North-Holland, 1977.
12. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.